v1.0

Sistemi Concorrenti e di Rete LS

II Facoltà di Ingegneria - Cesena

a.a 2008/2009

[module lab 1.2]
STRUCTURING PROGRAMS IN TASKS

STRUCTURING CONCURRENT PROGRAMS INTO TASKS

- *Task* concept
  - abstract, discrete, independent unit of work
  - **uncoupled from the notion of *thread***
- "Division of Labor" pattern (aka dividi-et-impera strategy)
  - indentify *task boundaries*
    - tasks as independent activities
    - not depending on state / results / side effects of other tasks
  - choose a *task execution policy*
    - objectives
      - good throughput
      - good responsiveness
      - gracefull degradation
- Outcome
  - simplify program organization
  - facilitates error recovery by providing natural transaction boundaries
  - promotes concurrency

EXECUTING TASKS SEQUENTIALLY

- SingleThreadWebServer example

```
class SingleThreadWebServer {

  public static void main(String[] args) throws IOException {
    ServerSocket socket = new ServerSocket(80);
    while (true) {
      Socket connection = socket.accept();
      handleRequest(connection);
    }
  }

}
```

- Problems
  - poor performance & throughput
  - poor responsiveness
  - poor resource usage

ONE-THREAD-PER-TASK POLICY

- Explicitly creating threads for task
  - unbounded thread creation
- ThreadPerTaskWebServer example

```
class ThreadPerTaskWebServer {

  public static void main(String[] args) throws IOException {
    ServerSocket socket = new ServerSocket(80);
    while (true) {
      final Socket connection = socket.accept();
      Runnable task = new Runnable(){
          public void run(){
            handleRequest(connection);
          }
        };
      new Thread(task).start();
    }
  }
}
```

ONE-THREAD-PER-TASK POLICY: PROS AND CONS

- Advantages
  - improved responsiveness
  - improved throughput
  - improved resource usage
- Disadvantages & drawbacks
  - thread lifecycle overhead
    - thread creation and teardown are not for free
  - resource consumption
    - active threads consume system resources, especially memory
  - stability
    - there is a limit on how many threads can be created
- Concurrency hazard
  - it may appear to work fine during prototyping and completely fail when deployed under heavy load

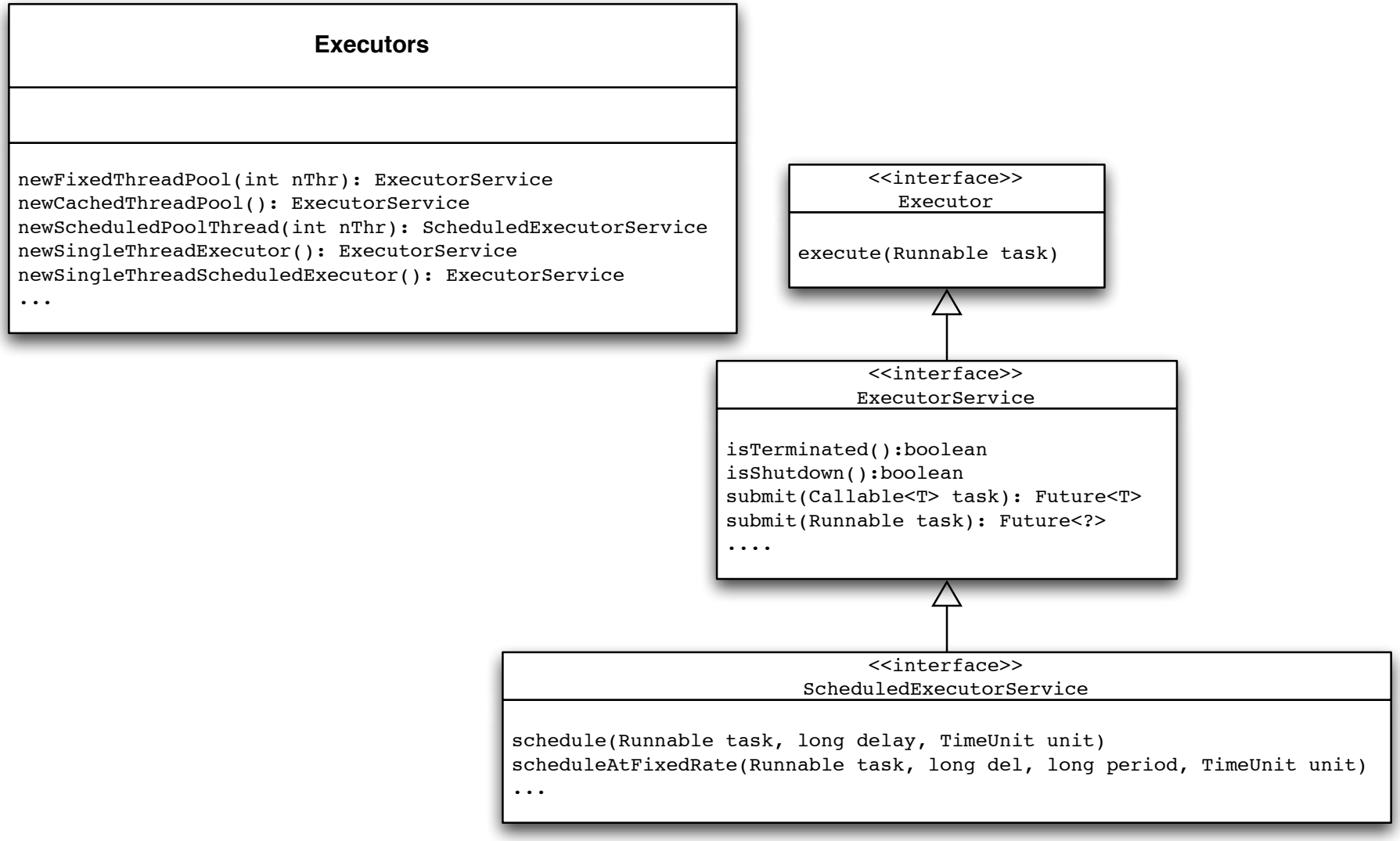
MORE FLEXIBLE POLICIES: EXECUTOR FRAMEWORK

- Task-oriented abstraction introduced with JDK 5.0
  - `java.util.concurrent`
- Direct support for decoupling *task submission* from *task execution*
  - task as logic unit of work
  - threads as the mechanism by which task can run asynchronously
- Producer-consumer pattern
  - the producers are the activities that submit tasks
  - the consumers are the threads that execute tasks

FRAMEWORK AND API

- **`Executor`** Interface
  - `execute` method to submit tasks
- Task described by **`Runnable`** interface
  - `run` method used to define task behaviour
- **`Executors`** class
  - utility for adding statistics gathering, application management and monitoring
  - factory for concrete available executors implementing specific *execution policy*

FRAMEWORK AND API



EXECUTION POLICIES

- Specifying "what, where, when and how" of task execution
  - when a task will be executed and by which thread?
  - in what order should tasks be executed (FIFO,LIFO,priority...)?
  - how many tasks may be executed concurrently?
  - If a task has to be rejected because the system is overloaded, which task should be selected as the victim?
  - what actions should be taken before or after executing a task?
- Resource management tool
  - the optimal policy depends on the available computing resources and QoS requirements
- Thread pool management
  - work queue
  - individual thread (consumer) behaviour
    - request next task todo
    - execute it
    - go back and wait for another task

EXECUTORS FACTORY: AVAILABLE EXECUTORS

- **`FixedThreadPool`**
  - one thread for each tasks up to the maximum pool size
  - keep the pool size constant
- **`CachedThreadPool`**
  - reap idle threads when pool size > current demand
  - create new threads if needed
  - no bounds
- **`SingleThreadExecutor`**
  - single worker, replacing it if dies unexpectedly
  - task guaranteed to be processed according to the task queue order (FIFO,LIFO,order)
- **`ScheduledThreadPool`**
  - fixed-size thread pool
  - support for delayed and periodic task execution

FIRST EXAMPLE: TaskExecutionWebServer

- Using the `FixedThreadPool` policy

```
class TaskExecutionWebServer {

  private static final int NTHREADS = 100;
  private static final Executor exec =
                          Executors.newFixedThreadPool(NTHREADS);

  public static void main(String[] args) throws IOException {
    ServerSocket socket = new ServerSocket(80);
    while (true) {
      Socket connection = socket.accept();
      Runnable task = new Runnable(){
          public void run(){
            handleRequest(connection);
          }
        };
      exec.execute(task);
    }
  }
}
```

POOL-BASED VS. THREAD-PER-TASK

- Better performance
- Strong impact on application stability
  - e.g.: a web server no longer fail under heavy load

EXECUTOR SERVICE

- Controlling Executor lifecycle
  - ExecutorService interface

```
public interface ExecutorService extends Executor {

  void shutdown();
  List<Runnable> shutdownNow();
  boolean isShutdown();
  boolean isTerminated();
  boolean awaitTermination(long timeout, TimeUnit unit)
                                    throws InterruptedException;
  ...

}
```

- Defining shutdown policy
  - from graceful shutdown to abrupt shutdown
  - challenging due to task asynchronous behaviour

EXECUTOR STATES

- **running**
  - when started
- **shutting down**
  - shutdown method is for *graceful* shutdown
    - no new tasks accepted
    - existing ones are allowed to complete
  - shutdownNow method is for abrupt shutdown
    - it attempts to cancel outstanding tasks
    - does not start any queued (not begun) tasks
  - reject execution handler
    - handling tasks submitted after shutdown
- **terminated**
  - once all tasks have completed

WebServerLifeCycle EXAMPLE

```
class WebServerLifeCycle {

  private static final ExecutorService exec = ...;

  public void start() throws IOException {
    ServerSocket socket = new ServerSocket(80);
    while (!exec.isShutdown()) {
      Socket conn = socket.accept();
      Runnable task = new Runnable(){
          public void run(){
            handleRequest(conn);
          }
        };
      try {
        exec.execute(task);
      } catch (RejectedExecutionException ex){
        if (!exec.isShutdown()){
          log("task submission rejected",ex);
        }
      }
    }
  }
  ...
```

WebServerLifeCycle EXAMPLE

```
  ...

  public void stop(){
    exec.shutdown();
  }

  void handleRequest(Socket connection){
    Request req = readRequest(connection);
    if (isShutdownRequest(req)){
      stop();
    } else {
      dispatchRequest(req);
    }
  }

}
```

EXECUTING DELAYED / PERIODIC TASKS

- Working with delayed and periodic tasks
  - scheduling the execution of tasks with absolute and relative timing
- **ScheduledThreadPoolExecutor** class
  - replacement for the Timer class

RESULT-BEARING TASKS: *CALLABLE*

- ExecutorService can manage “callable” tasks
  - tasks representing a deferred computation, completing with some kind of result
    - “`call`” instead of “`run`”

```
public interface Callable<V> {
  V call() throws Exception;
}
```

    - call method is meant to encapsulate the behaviour of the task doing some kind of computation and returning some kind V of value
  - submitted to an executor service by means of `submit` method, returning a **future**

```
public interface ExecutorService extends Executor {
  ...
  void execute(Runnable task);
  Future<T> submit(Callable<T> task);
  ...
}
```

FUTURES

- By submitting a task a *future* object is returned
  - provides methods to test whether the task has completed or been cancelled, to retrieve results or cancel the task

```
public interface Future<V> {
  boolean cancel(boolean mayIterruptIfRunning);
  boolean isCancelled();
  boolean isDone();
  V get() throws InterruptedException, ExecutionException,
                 CancellationException();
  V get(long timeout, TimeUnit unit) throws InterruptedException,
                          ExecutionException, CancellationException();
}
```